Rentify - Project Report

# Project Title

Rentify

# Group Members (fill names below)

1. Member 1: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ - Backend Lead  
2. Member 2: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ - Frontend Lead  
3. Member 3: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ - QA / PM  
4. Member 4: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ - DevOps & Documentation

# Chapter 1: Introduction

1.1 Project Summary  
Rentify is a web app where people can list items for rent and others can rent them. Owners add item details and pictures. Renters search, book, pay, and review. The backend uses Django and the frontend uses React with Vite.

1.2 Purpose  
To let people lend and borrow items easily, making it simple to earn and save money without buying things.

1.3 Objective  
- Let owners add and manage items.  
- Let renters search, book, and pay.  
- Keep bookings from clashing.  
- Support user accounts and reviews.  
- Make the app safe and easy to use.

1.4 Scope  
Included:  
- User signup/login, profile.  
- Item listing (with images), categories.  
- Search and basic filters.  
- Booking flow and booking status tracking.  
- Reviews and ratings.  
- Admin abilities through Django admin.

Not included (for now):  
- Production-grade payment flow (Razorpay keys are present in settings, but full flow may need wiring).  
- Mobile app (but site is responsive).  
- Full deployment automation (Docker/gunicorn/nginx not present).

1.5 Technology and short literature notes  
- Backend: Django 5.2, Django REST Framework, Simple JWT.  
- Frontend: React + Vite, Tailwind, Axios.  
- Database: Postgres configured in settings (local dev can use SQLite).  
- Storage: Firebase is used for image uploads.  
- Payments: Razorpay credentials expected via env vars.

# Chapter 2: Project Management

2.1 Project Planning  
Phases: planning (2 weeks), development (6 weeks), testing and deployment (2 weeks).

2.1.1 Development approach and justification  
Incremental development (Agile-like). Build core features first, then add extras.

2.1.2 Effort and cost (simple estimate)  
- Team: 4 people (Backend, Frontend, QA/PM, DevOps/Docs).  
- Time: ~10 weeks.

2.1.3 Roles and responsibilities  
- Backend: APIs, models, authentication.  
- Frontend: UI, routes, forms.  
- QA/PM: testing, user acceptance, coordinate tasks.  
- DevOps/Docs: deployment, environment setup, documentation.

2.1.4 Group dependencies  
- Frontend depends on backend API stability.  
- Payments depend on selecting and configuring a payment gateway.  
- Notifications depend on Firebase/email.

2.2 Project scheduling  
- Weeks 1–2: auth, basic item model, frontend skeleton.  
- Weeks 3–6: listing, searching, booking flows.  
- Weeks 7–8: payments, reviews, admin polish.  
- Weeks 9–10: testing, bug fixes, deploy prep.

# Chapter 3: System Requirements Study

3.1 User Characteristics  
- Owners: create item listings, set price and availability.  
- Renters: search, book, pay, review.  
- Admin: manage data via Django admin.

3.2 Hardware and Software Requirements  
Development:  
- Python 3.10+ and virtualenv.  
- Node.js (for frontend), Vite.  
- PostgreSQL for production; SQLite possible for quick dev.  
- Modern browser for UI.

Production (small):  
- Linux server with 1–2 vCPUs, 2–4 GB RAM, Postgres, object storage or Firebase, HTTPS.

3.3 Assumptions and Dependencies  
- Environment variables for email, Firebase, Razorpay must be set.  
- Users have internet access.  
- Third-party services (Firebase, Razorpay, email provider) are available.

# Chapter 4: System Analysis

4.1 Study of Current System  
- Auth: custom user model and email verification.  
- Items: categories, items, images, wishlist.  
- Rental: booking and booking history with price calc and statuses.  
- Review: rating code exists and is referenced by models.

4.2 Problems and Weaknesses  
- Payment flow may not be fully implemented.  
- Missing env vars will break email/Firebase/payments if not set.  
- Frontend serving via Django commented in settings.  
- Concurrency: availability checks exist but need DB-level locks for scale.

4.3 Requirements of New System  
- Functional: register/verify, login, item CRUD, image upload, search, booking, payment handling, reviews.  
- Non-functional: security (JWT), performance, maintainability.

4.4 Activity / Process  
- Owner creates item.  
- Renter requests booking.  
- System checks availability.  
- Booking created pending -> payment -> confirmed.

4.5 Integration  
- Firebase, email SMTP, Razorpay.

4.6 Features  
- Listings, calendar availability, booking, payments, reviews, admin.

4.7 Use Case and Models  
- Users, Items, Bookings, Reviews, Payments (expected).

# Chapter 5: System Design

5.1 Application Design  
- Input: HTTP API requests and files.  
- Output: JSON responses and emails.

5.1.1 Pseudocode (booking)  
- Check user auth.  
- Check item availability.  
- Calculate price in Booking.save().  
- Create booking pending.  
- Trigger payment flow.  
- On success: confirm booking.

5.2 API and Interfaces  
- Auth endpoints (register, verify, login, profile).  
- Item endpoints with image upload.  
- Booking endpoints.

5.3 Security  
- JWT auth, token blacklist, image validation, use HTTPS in production.

# Chapter 6: Implementation Planning

6.1 Environment  
- Activate virtualenv in project.  
- Install Python and Node dependencies.  
- Set env vars for email/Firebase/payments.

6.2 Module spec  
- Authentication app: register/login/verify/profile.  
- Items app: categories, items, images, wishlist.  
- Rental app: booking and history.  
- billing app: expected for payments.

6.3 Security Features  
- Password hashing, JWT, CSRF for forms, input validation, file size/type checks.

6.4 Coding Standard  
- Python: PEP8.  
- JS: ESLint.

# Chapter 7: Testing

7.1 Testing Plan  
- Unit tests for models and business logic.  
- API tests for endpoints.  
- Frontend component tests.

7.2 Strategy  
- Prioritize auth and booking flows.  
- Use CI for tests.

7.3 Test suites examples  
- Auth, Items, Booking, Review tests.

# Chapter 8: Conclusion and Discussion

8.1 Viability  
- Core features are implemented and project is viable.

8.2 Problems & Solutions  
- Double booking: add DB transactions and locks.  
- Missing env vars: provide .env.example and docs.  
- Image storage: consider S3 for production.

8.3 Summary  
- Backend is mostly complete; frontend skeleton exists. Next: payments, deployment, hardening.

# Chapter 9: Limitations and Future Enhancements

Limitations:  
- Payment flow may be incomplete.  
- No Docker or CI files present.

Future enhancements:  
- Implement payment gateway fully.  
- Add Docker and deploy pipeline.  
- Improve booking transaction handling and analytics.

# How to run locally

1. Activate virtualenv on Windows PowerShell: .\env\Scripts\Activate.ps1  
2. Install Python requirements: pip install -r requirements.txt (if present)  
3. Frontend: cd r\_frontend; npm install; npm run dev. Django: python manage.py migrate; python manage.py runserver  
5. Set env vars for email, Firebase, Razorpay before starting.